Edinburgh Napier University

SET08101 Web Tech

Lab 8 - Webapps Using Node.JS)

Dr Simon Wells

1 Aims

Our aim this week is to use JavaScript on the server side to generate our website from code. In previous weeks we've assumed that we know all the things that are going to happen, and everything that need to be displayed, at design time. However, quite often we will want to generate our HTML, perhaps based upon whether the user is logged in, or based upon a query supplied by the user. Over the next couple of weeks we'll add more features to this approach, such as using a datastore to hold the data for our web-app or using a RESTful approach to design our site. Before then, we need to see how to use JS to generate our website.

At the end of the practical portion of this topic you will:

- Install the Express framework & related supplementary Node packages
- Route HTTP requests in Node using the Express framework
- Generate an Express web-app and start using the Pug templating engine

2 Activities

2.1 The Express Framework

Express is a simple, minimal, and flexible web application framework for Node. It can be used to rapidly build robust and scalable web and mobile applications.

First we need to install the Express framework so that Node can use it. We'll use npm for that just like last week[1]. So navigate to the folder that holds node.exe then, using he Windows command line do:

```
> npm install express --save
```

After a few moments you should have the express framework and we can start using it. Create a folder, "helloexpress" then add a file called helloexpress.js into it. Edit helloexpress.js as follows:

```
1  var express = require('express');
2  var app = express();
3
4  app.get('/', function (req, res) {
5     res.send('Hello Express');
6  })
7
8  var server = app.listen(5000, "127.0.0.1", function () {
9     var host = server.address().address
10    var port = server.address().port
11
12    console.log("Listening on http://%s:%s", host, port)
13 })
```

We can now run that simple express web app as follows:

```
> node helloexpress.js
Listening on http://127.0.0.1:5000
```

[1] If you haven't completed the introduction to Node from last week then do that first otherwise the rest of this lab is just going to be increasingly confusing.

Note that the output from running this should be a message telling us that the web-app is running and suggesting a URL that it is available at. Note that 127.0.0.1 is an IP address that points to your local machine. If it doesn't work on your machine then you can also try `http://localhost:5000` and `http://0.0.0.0:5000` as there canbe some variation between machines, operating systems, and browser versions. All three addresses all basically mean the local machine however and are different ways of expressing the same concept.

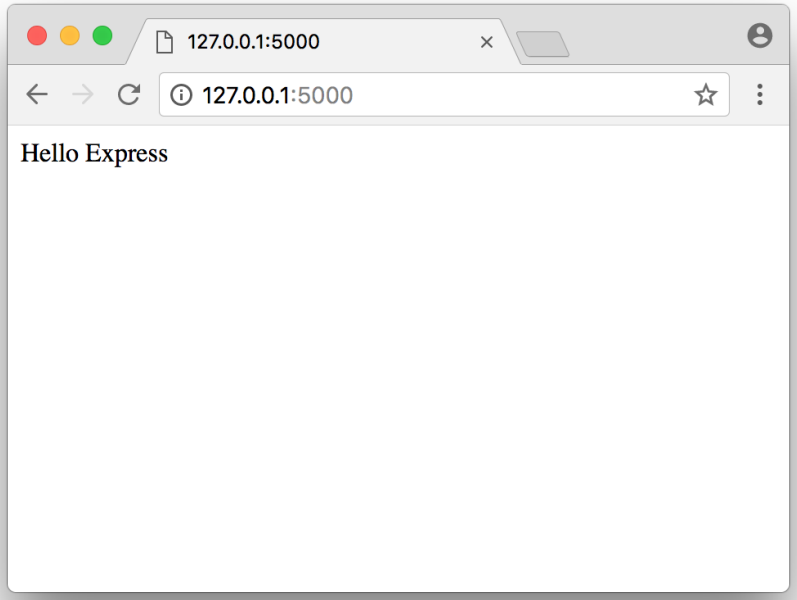

Figure 1: The output from the helloexpress.js web app

2.2 Routing in Express

Really all a web-app has to do is route messages around, for example, handle an incoming message to a given url and return the correct result (usually some HTML) so nearly all web-app frameworks are built around the concept of routing and Express is no different. What are the message that Express routes around? Basically they are objects reprepresenting core concepts of HTTP; requests and responses. Our web-app will recieve a request and return the correct response. Until now this has been straightforward because the correct response for a static web-server when it receives a request is to send back the resource that was requested. However with web-apps we intend to add some dynamic behaviour rather than merely returning static resources.

Express web-apps use *callback functions* to implement routing. The parameters passed to the callback functions are *request* and *response* obejects. This fragment gives you some indication of the structure of an express callback (we'll use it in earnest over the next few examples):

```
1 app.get('/', function (req, res) {
2    // We would include what the function does here
3 })
```

Notice that there are *req*uest and *res*ponse objects passed to the function as parameters. These directlt represent the incoming HTTP request and outgoing HTTP response, respectively. Something else to notice is that we can almost read the first line as follows" "the app object has a get method for the root (/) route that takes a request object, does some stuff and finally returns a response object". We can also interact with those req and res objects from code, for example we can print them to find out information about the incoming request or what information is stored by default in the response. We can of course also manipulate the response object. In fact, we've already seen a working example of a callback function in our helloexpress example earlier, e.g.

```
app.get('/', function (req, res) {
   res.send('Hello Express');
})
```

Note that in order to get "Hello Express" to be displayed in the browser we called the send() method of the response object. We can interact with the response object in a number of interesting ways.

For the moment though, let's consider one of the primary aspects of HTTP, particularly in relation to REST. That is, HTTP verbs. The different ways that a client can interact with an HTTP resource are primarily described by HTTP verbs. We've already seenand probably used thousands of time, the HTTP GET verb. This is the verb that is sent by default when a browser or other client interacts with an HTTP resource. Note that for the moment we'll consider an HTTP resource to be some data located at a given web address. We can interact with that resource at the given address in a whole bunch of different ways. For the moment let's consider the localhost "/" resource. Without doing anything cleve in JS or anything else, we can use different HTTP verbs to cause the server to do different things in relation to the resource just by calling the resource in different ways. For example:

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
   res.send('A GET request to the root resource');
})

app.post('/', function (req, res) {
   res.send('A POST request to the root resource');
})

app.head('/', function (req, res) {
   res.send('A HEAD request to the root resource');
})

var server = app.listen(5000, "127.0.0.1", function () {
   var host = server.address().address
   var port = server.address().port

   console.log("Listening on http://%s:%s", host, port)
})
```

We have a small problem though. Our browser doesn't, by default, allow us to easily call routes using verbs other than GET. We could make a POST request if there was an HTML button to press, but other than that we are a bit stuck. We can use others verbs from JS though, and there are lots of tools to let us use those verbs, but browsers aren't one of them. We should consider why this is. Mostly it is because the web, as supported by HTTP, is about more than just accessing data from a web browser. There are many web clients that aren't browsers, for example, mobile applications, really anything that can consume an HTTP API.

One such tool is cURL, a command line application that is installed by default on many Linux and Mac machines and which is an easy download away for Windows users. Download the latest zip file from `https://curl.haxx.se/download.html` then unzip it somewhere useful. Note that the archives at the top of that page refer to the source code for the program. You should navigate down to the "Win32 Generic" section and download the latest zip from there[2]. You'll then need to navigate to that location using the Windows command line and find curl.exe which was located in the src folder on the build that I downloaded.

We can now use cURL to call our various routes, for example, calling the GET route:

```
> curl.exe http://localhost:5000
A GET request to the root resource
```

[2]I downloaded the following when testing things out recently: `https://dl.uxnr.de/build/curl/curl_winssl_msys2_mingw32_stc/curl-7.58.0/curl-7.58.0.zip`

Notice how the output is similar to what we would get in the browser. We can now also call the POST route:

```
> curl.exe http://localhost:5000 -X POST
A POST request to the root resource
```

This time the output is different because we couldn't issue a POST request directly from the browser without implementing a button on a web page to issue the POST. However, cURL lets us issue the request directly. Let's try the PUT verb:

```
> curl.exe http://localhost:5000 -X PUT
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8"
<title>ERROR</title>
</head>
<body>
<pre>Cannot PUT /</pre>
</body>
</html>
```

Ahhh. This is different. We got an error. This is because we haven't implemented a handler for PUT requests to this route. Instead, when our node app receives the request an error page is automatically generated in HTML.

Finally, let's call that HEAD route and see what happens:

```
> curl.exe http://localhost:5000 --head
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 34
ETag: W/"22-4d0H1eyiftrrcLVM4PJK+kdQ6Nc"
Date: Mon, 26 Feb 2018 17:15:03 GMT
Connection: keep-alive
```

Hmmmm. That's different again. Firstly, we had to call this verb using the –head argument instead of using -X head as we would for PUT, POST, etc. Additionally, we on't see the actual content of the response from the / route when we use the HEAD command. Instead we just get the header information. That's the point, HEAD is really useful for getting the information that the server will send back from a route but ignoring the actual body of the document, i.e. the HTML document.

Express supports a whole bunch of routing methods that correspond to HTTP verbs, for example:

- checkout
- copy
- delete
- get
- head
- lock
- merge
- mkactivity
- mkcol
- move
- m-search
- notify
- options
- patch
- post
- purge
- put
- report
- search
- subscribe
- trace
- unlock
- unsubscribe

This is an opportunity to do some background reading. First have a look at tthe Express documentation[3] then perhaps investigate HTTP verbs; the Mozzila developer documentation is a good place to start[4].

[3] https://expressjs.com/en/api.html
[4] https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

2.3 Raw HTML from Express

So far we've only returned plain text from Express. What we probably want to do is to respond with an HTML document. We can do that in code, or using a templating library. Let's start with generating some raw HTML in our response:

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
    res.set('Content-Type', 'text/html');
    res.send(new Buffer('<h1>The root route</h1>'));
   })

app.get('/hello', function (req, res) {
    res.set('Content-Type', 'text/html');
    res.send(new Buffer('<h2>The hello route</h2>'));
   })

app.get('/goodbye', function (req, res) {
    res.set('Content-Type', 'text/html');
    res.send(new Buffer('<p>The <b>goodbye</b> route</p>'));
    })


var server = app.listen(5000, "127.0.0.1", function () {
   var host = server.address().address
   var port = server.address().port

   console.log("Listening on http://%s:%s", host, port)
})
```

We'll revist this later because we can use a similar approach to return JSON instead of HTML. That said, Express has a much better way to deal with HTML, using templates.

2.4 Using Templates from Express

Express uses template engines to control the generation of HTML so that we don't have to do everything from scratch. Instead we create templates that Express processes to generate the HTML documents that are returned to the client.

To use templates with Express we need to install some extra packages using NPM. First Pug which is a templating engine:

```
> npm install pug --save
```

We also need the express-generator package which gives us a way to generate the skeleton of an express web-app:

```
> npm install express-generator -g
```

Now we can use the express generator to create our skeleton. We'll also specify to use pug and give the name of our new app as helloapp

```
> express --view=pug helloapp
```

This will generate a structure containing bin, public, routes, and views folders as well as an app.js file and a package.json file. Explore the various folders and files. Notice that views contains a number of files that end in '.pug' which are the tempaltes for your HTML pages.

Figure 2: The structure of the Express web app generated by express-generator.

Once a skeleton is generated we can use npm to install all the Node modules that our new express app needs, e.g.

```
> npm install
```

This will cause a folder called node_modules to be created and a bunch of standard packages to be installed into it. You should be getting the idea now that Node likes to break things down into lots of smaller packages then give you tools to help manage them all.

This should be enough set up to enable us to get a basic express app running. One last small thing to do if we're on a JKCC lab machine is to alter the port number that our app will run on. The default is 3000 and we have 5000 reserved for use on this module. So open the file called www in the folder called bin and change the following line:

```
1 var port = normalizePort(process.env.PORT || '3000');
```

to:

```
1 var port = normalizePort(process.env.PORT || '5000');
```

Now let's run our app[5]:

```
> set DEBUG=myapp:* & npm start
```

We should now be able to access our app in our browser by visiting `localhost:3000`.

To add additional routes to our web-app, we merely add them to the routes/index.js file. This should look familiar because it's similar to what we were editing earlier when we were using Node without the express framework. This is index.js:

```
1 var express = require('express');
2 var router = express.Router();
3
4 /* GET home page. */
5 router.get('/', function(req, res, next) {
6   res.render('index', { title: 'Express' });
7 });
8
9 module.exports = router;
```

We can add whichever routes we want to *routes/index.js* like so:

```
1  var express = require('express');
2  var router = express.Router();
3
4  /* GET home page. */
5  router.get('/', function(req, res, next) {
6    res.render('index', { title: 'Express' });
7  });
8
9  router.get('/hello', function(req, res, next) {
10   res.render('index', { title: 'The Hello Route' });
11 });
12
13 module.exports = router;
```

We can also add extra templates as required into the views folder. These all use the Pug language instead of HTML. For example, the template used by both the index and hello routes above is the index.pug file which looks like this:

```
1 extends layout
2
3 block content
4   h1= title
5   p Welcome to #{title}
```

Notice that both routes are using the same template, the idea is to capture the common aspects of the different HTML pages that will make up our site. This way we can have a small number of templates, but perhaps hundreds or even thousands of pages can use the same template. By suppling different data to the template from the route function we get the tempalte to be completed in different ways which leads to different HTML being generated and returned to the browser. For example, our root and hello functions in routes/index.js are passing different data to the template so that the title parameter is compelted differently. Using templates make it even more important to think about the layout of your site before you try to build it. A rough preliminary design will give your some idea of how many templates are needed and this number should, as a rule be much lower than your total number of HTML pages. If you have as many templates as HTML pages then you're doing it wrong.

[5]If you're on Mac or Linux then do this instead: $ DEBUG=myapp:* npm start: